CS 330 – Spring 2023 – Homework 03

Due: Wednesday 2/15 at 11:59 pm on Gradescope

Collaboration policy Collaboration on homework problems is permitted, you are allowed to discuss each problem with at most 3 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (these include anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise description of the algorithm in English and pseudocode (*),
2. a proof of correctness,
3. an analysis of the asymptotic running time of your algorithm.

You may use anything we learned in class without further explanation. This includes using algorithms from class as subroutines, stating facts that we proved in class, e.g. correctness of subroutines, running time of subroutines and use the notation. Your description should be at the level that it is clear to a classmate who is taking the course with you.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

(*) It is fine if the English description concentrates on the high level ideas and doesn't include all the details. But the reader should not have to figure out your solution solely based on the pseudocode. You can also add comments to your pseudocode, in fact that is best practice.

- By: Gianna Sfrisi U77992006
- Collaborators: Albert Slepak U00956163

Problem 1. (Shortest building time) Imagine you and your friends are working together to cook an extremely complicated recipe for an upcoming potluck. Every one of your friends has a unique job within this process, and you may assume for simplicity that every job takes exactly one hour. You want to get this done as soon as possible, but some jobs need to be done before others, while some jobs can be done simultaneously. For example, chopping the vegetables can be done while someone prepares pots and pans, but the food needs to be cooked before it is plated.

1. Describe how you can represent this problem using a directed graph. Make sure to give a precise description of how nodes, edges and directions are assigned.

   One could represent this problem using a directed graph to show which steps are prerequisites for steps later in the graph. The first nodes in the graph (the "head" nodes) would be the steps that could take place at any time with no prior steps. These nodes would only have outward-directed edges and no inward-directed edges. The middle nodes in the graph would have a combination of both inward-directed and outward-directed edges. This is due to the fact that there are steps that must be completed prior to this task but there are also nodes that cannot be reached until after these steps are completed. The last type of node in this graph would be the ones that have only inward-directed edges. These steps are the ones in the final part of the recipe that have no steps after it is completed.

   One real-life example of this is if someone needed to 1) buy carrots, 2) cut the carrots and lastly 3) cook the carrots. The carrots must be purchased before they are cut or cooked. Therefore the step of buying the carrots would be considered the "head" node. The middle nodes would be represented by cutting the carrot; since the is at least one step (or "node") before it and at least one node after it. The final node would be cooking the carrot because after that is completed there are not more steps and the process (or graph) is completed.

2. You want to find out whether it is even possible to cook the recipe. For this you need to know that there are no two jobs that mutually depend on each other. (e.g. j_1 needs to be finished before j_2, but j_2 needs to be finished before j_1). Formulate what property your graph needs to have so that this doesn't happen. Give a one sentence explanation why.

   The graph must be a DAG; cycles represent two steps that depend on each other which would make it impossible to finish the job.

3. Give an algorithm that receives as input a list of m dependencies among n tasks (in the form of pairs (i, j) where task i must be performed before task j), and outputs a schedule that accomplishes all the tasks in as few hours as possible. Your algorithm should output, for each task, the hour on which it should be performed.

For example, suppose that we have six tasks a, b, c, d, e, f with seven dependencies

$$(a, b), (a, c), (a, d), (b, f), (c, f), (d, f), (e, f).$$

Then the following outputs would be acceptable:

Task	hour
a	1
b	2
c	2
d	2
e	1
f	3

or

Task	hour
a	1
b	2
c	2
d	2
e	2
f	3

For full credit, your algorithm should run in time $O(m + n)$. You may use algorithms from lab or the textbook (Chapter 3).

Algorithm:

```
function traverseNode(digraph, node, hour) {
        /*Keep the higher value for hour (must take every path into account) */
        node.hour = max(n.hour, hour);

        if (digraph.has(node)):
                for (child in digraph[node]) do
                        traverseNode(digraph, child, hour + 1);
                done
}
headNodes = [] /* list of nodes that can occur in the first hour */
/* Construct a digraph given the current dependencies */
for (nodeBefore, nodeAfter) in dependencies do
        digraph[nodeBefore].append(nodeAfter);
done
parents, dist = DFS(digraph); /*Calling DFS (reference from lecture) on the digraph */
/* Find all "head" nodes */
for node in digraph do
        if dist[node] == 0:
                headNodes.append(node)

/* Traverse through each node in the digraph, assign headnodes with a value of 1*/
/*Assign other nodes based on the order they are found*/
for (head in headNodes) do
        // Recursively assign task hours
```

```
        traverseNode(digraph, head, 1);
done
/* Print out the results from the algorithm*/
        for node in digraph:
                print(node " must be done at hour " value)
```

Proof Of Accuracy:
In order to start, the algorithm creates a digraph given the list of dependencies. The algorithm then calls the original DFS algorithm from lecture on the digraph in order to find the head nodes of the graph. The head nodes can all occur in the first hour since there are no tasks that must be completed beforehand. The algorithm then uses a loop to cycle through the neighbors of the given head node, assigning a value based on the time the neighbor had been discovered. Once the algorithm starts connecting nodes that have previously found neighbors, it compares the original value assigned and the new value to see which is the higher. If there are multiple paths to a node the algorithm will assign the node with the higher value because all neighbors and paths must be taken accounted for. Once every edge has been iterated over and every node has been assigned a value, the algorithm prints out a series of statements of which hour every task should be completed.

Run-time Analysis:
The run-time is $\Theta(n + m)$. Constructing the digraph at the initial part of the algorithm runs at $\Theta(m)$ as it must run through every dependency listed in the instructions of the problem. Calling DFS (referencing from lecture) on the now constructed digraph runs at time $\Theta(n+m)$. After running DFS, the algorithm must identify which nodes have a distance of 0 (aka a "head" node) and assign those nodes with a value of 1. Since the algorithm must iterate through every node to identify which have a distance of 0, this step has a run time of $\Theta(n)$. The final part of the algorithm that traverses through every node and possible neighbor, assigning values along the way. This step runs at a time of $\Theta(n+m)$ as each node and each of their neighbors must be accounted for. Hence the total run time for this algorithm would be $\Theta(3n+3m)$, however, since constants do not have a strong effect on run-time, the algorithm would be considered to have a run-time of $\Theta(n + m)$.

Problem 2. (Directing edges) You are given an *undirected* graph $G(V, E)$. Design an efficient algorithm *Directions*( ) that can decide whether it is possible to assign directions to the edges such that there is no directed path of length 2 or higher in the graph. That is, every node has either zero indegree or zero outdegree.

The algorithm *Directions*( ) should take as input the adjacency list of graph G (you may assume it is the hash table implementation). It should either return the correctly directed graph (it can be in the form of the adjacency list of a directed graph) or return "not possible to direct".

1. Look at examples of graphs in Figure 1 and try to assign the directions. (You can draw a few examples yourself too) Based on your observations, formulate for what type of graphs it is possible to assign the directions as desired.

   Your answer should be in the form "Given the graph $G(V, E)$ we can assign directions to the edges in E if and only if ...". (Your answer should be a clear and concise description of your observation. Probably just one or two sentences. You should NOT hand in the solution for the example graphs.)

   Given the graph $G(V, E)$ we can assign directions to the edges in E if and only if the nodes of the graph do not create a triangular shape where all three nodes are connected, therefore creating a cycle.

2. Find and analyze the algorithm *Directions*( ).

   Algorithm:
   /* G is hash table, the adjacency list of a graph */
   /* Let s be a random source node */
   resultingNodes = {} /* the list that will be returned at the end (empty to start)*/
   /* v represents each node in G */ /* u represents each edge of node v */
   /* Iterates through the list of head nodes*/
   /* For each neighbor add edges based on the current in or out degree*/

   /* Checks the in and out degree of each node, assign edges accordingly */

```
for v in G:
    for u in G[v]:
        if v has an in(deg) > 0 && out(deg) > 0
            return "not possible to direct"
```

   /* Adds edges in the direction based on the nodes current degree */

```
        elif v has a in(deg) > 0
            G[v].add(u)
        elif v has a out(deg) > 0
            G[u].add(v)
    return G
```

Proof Of Accuracy:
In order for the graph to be considered a directed graph, every edge must be assigned a direction. Following the guidelines of the problem, no nodes can have edges in the inward and outward direction simultaneously; additionally, there cannot be a path of a length longer than or equal to two. This rule automatically removes the possibility of any cycles in the graph. Hence the returned graph, if possible, will always be a DAG. In the algorithm, this is immediately searched for in the first line of the for loop as if a node has an inward edge and an outward edge, there automatically is a path of length two and a possibility for a cycle. The next part of the algorithm assigns a directed edge between the current node s and its neighbors. The direction is assigned based on what other connections that node already has with previous neighbors. The loop then cycles through every node and returns the newly directed graph.

Run-time Analysis:
The run-time of this algorithm is $\Theta(n+m)$. The algorithm cycles through every possible node in the for loop which would run at $\Theta(n)$. Inside the for loop, the algorithm creates a directed edge between the current node and its neighbors, hence looping through every possible edge to assign a direction. Since the algorithm runs through both every possible node and every edge the total run time would be $\Theta(n+m)$.

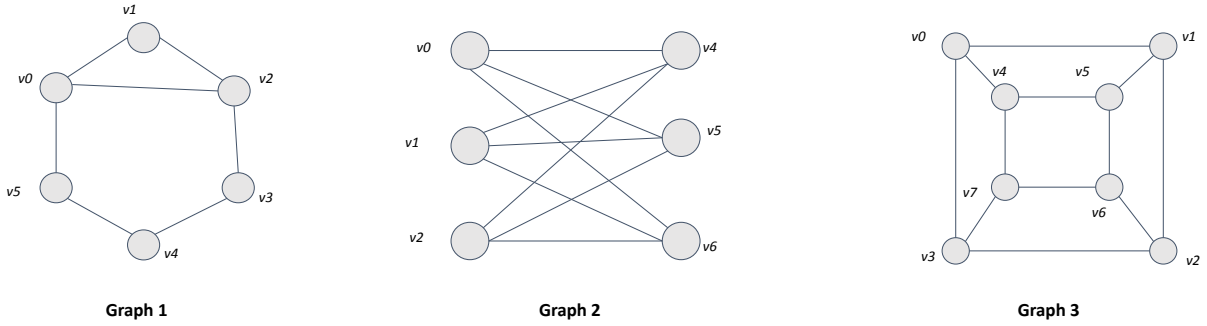

Figure 1: Can these three graphs be made into directed graphs with no directed path of length 2 or above?

For the record, Graph 1 cannot be a directed graph due to the connection between v0 and v2; a cycle is created between v0, v1, and v2 and it would be impossible to assign a direction to all three edges combining those nodes without creating a path longer than two. However, it is possible for Graph 2 and Graph 3 to be assigned directions for every node.